CHAPTER 6: STRUCTURES

A *structure* is a collection of one or more objects, probably of different types, grouped together under a single name for convenient handling. (Structures are called "records" in some languages, most notably Pascal.) The elements of a structure are called *members.*

The classical example of a structure is the payroll record: an "employee" is described by a set of attributes which would normally be separate variables, such as name, address, social security number, salary, etc. Some of these variables in turn could be structures: a name has several components, as does an address and even a salary.

If well used, structures contribute markedly to orderly data layouts. Large programs in particular benefit from the organizing force that structures provide. In this chapter we will try to illustrate how structures are used. The programs we will use are bigger than many of the others in the book, but still of modest size.

6.1 Basics

To illustrate the basic syntactic rules, let us revisit the date conversion routines of Chapter 5. A date clearly consists of several pieces — a year, a day of the year, a month, and a day of the month. So these could all be placed into a single structure like this:

```
struct date {
        int     year;
        char    leap;
        int     yearday;
        int     month;
        char    mon_name[4];
        int     day;
};
```

We include **leap** and **mon_name** to show a structure containing several types of variables.

The keyword **struct** introduces a structure declaration, which is a list of variable declarations enclosed in braces. The format of the declaration is free, but conventionally declarations are written one variable per line. An optional name called the *structure tag* may follow the word **struct** (as with **date** here). The tag names this kind of structure, and can be used subsequently as a shorthand for the whole declaration. The names used within the structure declaration are called "members."

Although a structure member and an ordinary variable can have the same name without conflict, this can be confusing unless used carefully.

The right brace may be followed by a list of variables, just as for any basic type. That is,

```
struct { ... } x, y, z;
```

is syntactically equivalent to

```
int x, y, z;
```

in the sense that each statement declares x, y and z to be objects of the named type.

A structure declaration that is not followed by a list of variables allocates no storage; it merely defines a *template* or shape of structure. If the declaration is tagged, however, the tag can be used later to declare actual instances of the structure. For example, given the declaration of **date** above,

```
struct date   birthdate;
```

makes a variable **birthdate** which is a structure of type **date**.

An external or static structure can be initialized by following its declaration by a list of initializers for the components:

```
struct date birthdate = {1776, 1, 186, 7, "Jul", 4};
```

describes the date of independence of one of the larger American republics.

Now we can compute. A member of a structure is referenced in an expression by a construction of the form

structure . member

To set **leap** in the structure **d**, we write

```
d.leap = (d.year%400 == 0) || (d.year%100 != 0 && d.year%4 == 0);
```

Structures can be nested; a payroll record might actually look like

```
struct person {
        char   name[50];
        char   address[50];
        int    zipcode;
        long   ss_number;
        float  salary;
        struct date   birthdate;
        struct date   hiredate;
};
```

The **person** structure contains two dates. To refer to the month of birth for someone,

```
struct person p;

        m = p.birthdate.month
```

And to access the first character of the **name** field in this structure, write

```
p.name[0]
```

6.2 Structures and Functions

There are a number of restrictions on C structures. The essential rules are that the only things that you can do a structure are to take its address with &, and access one of its members. This implies that structures may not be assigned to or copied as a unit, and that they can not be passed to or returned from functions. Pointers to structures do not suffer these limitations, however, so structures and functions do work together comfortably. Finally, automatic structures, like automatic arrays, cannot be initialized; only external or static structures can.

Let us investigate some of these points by rewriting the date conversion functions of the last chapter to use structures. Since the rules prohibit passing a structure to a function directly, we must either pass the components separately, or pass a pointer to the whole thing. Since structure members are just ordinary variables, the first alternative is in effect what we've already done:

```
d.yearday = day_of_year(d.year, d.month, d.day);
```

So we might better pass the structure pointer. If we have

```
struct date   hiredate;
```

we can then write

```
yearday = day_or_year(&hiredate);
```

to pass the pointer to **day_of_year**. The function itself has to be modified too, since its argument is now a structure pointer rather than a list of variables.

```
day_of_year(pd)     /* return day of year */
struct date *pd;
{
        int i;

        pd->leap = (pd->year%400 == 0)
         || (pd->year%100 != 0 && pd->year%4 == 0);
        for (i = 1; i < pd->month; i++)
                pd->day += day_tab[pd->leap][i];
        return(pd->day);
}
```

The declaration

```
struct date *pd;
```

says that **pd** is a pointer to that kind of structure. The notation exemplified by

```
pd->leap
```

is new. If **p** is a pointer to a structure, then

p -> *member of structure*

refers to the particular member. So

```
pd->leap
```

is the leap year indicator,

```
pd->year
```

is the year, and so on. Structure pointers are so common that the -> notation is provided as a shorthand for the equivalent

```
(*pd).year
```

As you might expect, there is considerable potential for error in coupling a pointer to one structure with a member of another.

For completeness, here is the other function, **month_day**, rewritten with structures. The only change necessary is to convert references to arguments into structure offsets.

```
month_day(pd)       /* set month and day from day of year */
struct date *pd;
{
        int i;

        pd->leap = (pd->year%400 == 0)
          || (pd->year%100 != 0 && pd->year%4 == 0);
        pd->day = pd->yearday;
        for (i = 1; pd->day > day_tab[pd->leap][i]; i++)
                pd->day -= day_tab[pd->leap][i];
        pd->month = i;
}
```

For complicated pointer expressions, it's wise to use parentheses to make it clear who goes with what. For example, given the declaration

```
struct {
        int x;
        int *y;
} *p;
```

then

```
p->x++          increments x
++p->x          so does this
(++p)->x          increments p before getting x
*p->y++         uses y as a pointer, then increments it
*(p->y)++         so does this
*(p++)->y         uses y as a pointer, then increments p
```

The way to remember these is that ->, . (dot), ( ) and [ ] bind very tightly. An expression involving one of these is treated as a unit: `p->x`, `a[i]`, `y.x` and `f(b)` are names exactly as `x` is.

6.3 Arrays of Structures

Suppose we want a program that counts occurrences of each C keyword. One possibility is to use two parallel arrays **keyword** and **keycount** to store a pointer to the name and the count. But the very fact that the arrays are parallel indicates that a different organization is possible. Each keyword entry is really a pair:

```
char  *keyword;
int   keycount;
```

and there is a whole set of pairs.

The structure declaration

```
struct key {
        char *keyword;
        char keycount;
} keytab[NKEYS];
```

defines an array **keytab** of structures of this type, and allocates storage to them. Since this structure actually contains a constant set of names, however, it is easy to initialize it once and for all when it is declared. The structure initialization is quite analogous to earlier ones — the declaration is followed by a list of initializers enclosed in braces:

```
struct key {
        char  *keyword;
        int   keycount;
} keytab[ ] ={
        "break",        0,
        "case",         0,
        "char",0,
        "continue",     0,
        "default",      0,
        "do",   0,
        /* ... */
        "while",        0,
        NULL, 0
};
```

The declarations are listed in pairs corresponding to the structure members. It is more precise to enclose initializers for each "row" or structure in braces, as in

```
        { "break", 0 },
        { "case", 0 },
        ...
```

but the braces may be omitted when the initializers are simple and there are no omissions. We terminated the list with a null pointer to make it easy for programs to find the end of the array. As usual, the compiler computes the size of the array **keytab** itself when initializers are present and the [ ] is left empty.

The counting program can now be finished. To illustrate communication with external structure definitions, we have chosen to modify the binary search program so that the array it searches is an external structure, rather than an argument. This

degeneralizes it, of course, but in this case it is actually the easiest way to do the job. We must also alter the binary search routine to compare strings of characters instead of integers.

```
struct key {
        char  *keyword;
        int    keycount;
} keytab[ ] ={
        "break",      0,
        "case",       0,
        /* ... */
        "while",      0,
        NULL, 0
};

#define     MAXWORD  20

main( )     /* count occurrences of C keywords */
{
        int n, t;
        char word[MAXWORD];

        while ((t = getword(word, MAXWORD)) != EOF)
                if (t == LETTER)
                        if ((n = binary(word)) >= 0)
                                keytab[n].keycount++;
        for (n = 0; n < NKEYS; n++)
                if (keytab[n].keycount > 0)
                        printf("%4d %s\n",
                          keytab[n].keycount, keytab[n].keyword);
}
```

--

```
binary(word) /* find word in keytab[0] ... keytab[NKEYS-1] */
char *word;
{
        int low = 0, high = NKEYS-1, mid, cond;

        while (low <= high) {
                mid = (low+high) / 2;
                if ((cond = strcmp(word, keytab[mid].keyword)) == 0)
                        return(mid);
                else if (cond < 0)
                        high = mid - 1;
                else
                        low = mid + 1;
        }
        return(-1);
}
```

We will describe the function **getword** in a moment; for now it suffices to say that it returns **LETTER** each time it finds a word, and copies the word into its first argument.

The quantity **NKEYS** is the number of keywords in **keytab**. Although we could count this by hand, it's a lot easier to do it by machine. One possibility would be to write a loop which counts along **keytab** until it finds the null pointer, then assign it to a global variable.

But this is more than is needed, since the size of the array is completely determined at compile time. C provides a compile-time unary operator called **sizeof** which can be used to compute the size of any object. The expression

sizeof (*object*)

yields an integer equal to the size of the specified object. (The size is given in unspecified units called "bytes," which are generally the same size as a **char**.) The object can be an actual variable or array or structure, or it can be a type like **int** or **double**. In our case, the number of keywords is

sizeof (keytab) / sizeof (individual entry)

Thus by including in the program the definition

```
#define     NKEYS     (sizeof (keytab) / sizeof (struct key))
```

NKEYS is defined properly.

Now the function **getword**. We have actually written a more general **getword** than is necessary for this program, but it is not really more complicated. **getword** returns the next "token" from the input, where a token is either an identifier in the C sense (that is, a maximal string of letters and digits beginning with a letter), or a single character. The type of the object is returned as a function value; it is **LETTER** if the token is a word, **DIGIT** for a digit, or the character itself if it is non-alphanumeric.

```
getword(w, lim)      /* get next word from input */
char *w;
int lim;
{
        int c;

        if (type(c = *w++ = getc( )) != LETTER) {
                *w = '\0';
                return(c);
        }
        while (--lim > 0)
                if ((c = type(*w++ = getc( ))) != LETTER && c != DIGIT)
                        break;
        ungetc(c);
        *(w-1) = '\0';
        return(LETTER);
}
```

Notice that **getword** uses the routines **getc** and **ungetc** which we wrote in Chapter 4: when the collection of an alphabetic token stops, **getword** has gone one character too far. The call to **ungetc** pushes that character back on the input for the next call.

getword calls **type** to determine the type of each individual character of input.

```
type(c)      /* return type of character */
int c;
{
        if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
                return(LETTER);
        else if (c >= '0' && c <= '9')
                return(DIGIT);
        else
                return(c);
}
```

The symbolic constants **LETTER** and **DIGIT** can have any values that do not conflict with non-alphanumeric characters and **EOF**; The obvious choice is

```
#define     LETTER     'a'
#define     DIGIT '0'
```

Exercise 6-1: **getword** can be faster if calls to the function **type** are replced by references to an appropriate array **type[ ]**. Make this modification, and measure the change in speed of the program. □

6.4 Pointers to Structures

To illustrate some of the considerations involved with pointers and arrays of structures, let us write the keyword-counting program over again using pointers instead of array indices.

The external declaration of **key** need not change, but **main** and **binary** do need modification.

```
main( )       /* count occurrences of C keywords; pointer version */
{
        int n, t;
        char word[MAXWORD];
        struct key *binary( ), *p;

        while ((t = getword(word, MAXWORD)) != EOF)
                if (t == LETTER)
                        if ((p = binary(word)) != NULL)
                                p->keycount++;
        for (p = keytab; p < keytab + NKEYS; p++)
                if (p->keycount > 0)
                        printf("%4d %s\n", p->keycount, p->keyword);
}

struct key *binary(word)   /* find word in keytab[0] ... keytab[NKEYS-1] */
char *word;
{
        int cond;
        struct key *low = keytab, *high = keytab+NKEYS-1, *mid;

        while (low <= high) {
                mid = low + (high-low) / 2;
                if ((cond = strcmp(word, mid->keyword)) == 0)
                        return(mid);
                else if (cond < 0)
                        high = mid - 1;
                else
                        low = mid + 1;
        }
        return(NULL);
}
```

There are several things worthy of note here. First, the declaration of **binary** must indicate that it returns a pointer to the structure type **key**, instead of an integer; this is declared both in **main** and in **binary**.

Second, all the accessing of elements of **keytab** is done by pointers. This causes one significant change in **binary**: the computation of the middle element can no longer be simply

```
mid = (low+high) / 2
```

because the *addition* of two pointers will not in general produce any kind of a useful answer (even when divided by 2). This must be rearranged into

```
mid = low + (high-low) / 2
```

which increments low by the right number of elements.

You should also study the initializers for **low** and **high**. It is possible to initialize a structure pointer to the address of a previously defined object; that is precisely what we have done here.

Finally, in **main** we wrote

```
for (p = keytab; p < keytab + NKEYS; p++)
```

We could equally well have used the fact that **keytab** is terminated by a null pointer, and written

```
for (p = keytab; p->keyword != NULL; p++)
```

If **p** is a pointer to a structure, any arithmetic on **p** takes into account the actual size of the structure. For instance, **p++** increments **p** by the correct amount to get the next element of the array of structures. But don't assume that the size of a structure is the sum of the sizes of its members — because of alignments of different sized objects, there may be "holes" in a structure.

6.5 Nested Structures

Suppose we want to handle the more general problem of counting the occurrences of *all* the words in some input. Since the list of words isn't known in advance, we can't sort it and use a binary search. Yet we can't do a linear search for each word as it arrives; the program would take forever. How can we organize the data to cope efficiently with a list of arbitrary words?

One solution is to keep the set of words sorted at all times, by placing each word into its proper position in the order as it arrives. This can't be done by shifting words in a linear array, though — that also takes too long. Instead we will use a data structure called a binary tree.

The tree contains one "node" per distinct word; each node contains

a pointer to the actual word
the count
a pointer to the left child
a pointer to the right child

No node may have more than two children; it might well have only zero or one.

The nodes are maintained so that at any node the left subtree contains only words which are less than the word at the node, and the right subtree contains only words that are greater. The tree is inherently recursive, of course, so recursive routines for insertion and printing will be most natural.

Going back to the description of a node, it is clearly a structure with four components:

```
struct node {        /* the basic node */
        char  *word;/* points to the text */
        int     count;
        struct node  *left;  /* left child */
        struct node  *right; /* right child */
};
```

This "recursive" definition of a node might look chancy, but it's actually quite correct. It is illegal for a structure to contain an instance of itself, but

```
struct node  *left;
```

is a *pointer* to a node, not a node itself.

The code for the whole program is surprisingly small, given a handful of supporting routines that we have already written. These are **getword**, to fetch each input word, and **alloc**, to provide space for squirreling the words away.

```
#define     MAXWORD  20

main( )     /* word frequency count */
{
        struct node *top, *tree( );
        char word[MAXWORD];
        int t;

        top = NULL;
        while ((t = getword(word, MAXWORD)) != EOF)
                if (t == LETTER)
                        top = tree(top, word);
        treeprint(top);
}
```

```
struct node *tree(p, w)     /* install w at or below p */
struct node *p;
char *w;
{
        struct node *talloc( );
        char *alloc( );
        int cond;

        if (p == NULL) {    /* a new word */
                p = talloc( );/* make a new node */
                p->word = alloc(strlen(w)+1);
                strcpy(p->word, w);
                p->count = 1;
                p->left = p->right = NULL;
        } else if ((cond = strcmp(w, p->word)) == 0)      /* repeated word */
                p->count++;
        else if (cond < 0)  /* lower goes into left */
                p->left = tree(p->left, w);
        else            /* greater into right */
                p->right = tree(p->right, w);
        return(p);
}

treeprint(p)  /* print tree p recursively */
struct node *p;
{
        if (p != NULL) {
                treeprint(p->left);
                printf("%4d %s\n", p->count, p->word);
                treeprint(p->right);
        }
}
```

The main routine simply reads words with **getword** and installs them in the tree with **tree**.

tree itself is straightforward. A word is presented by **main** to the top level (the root) of the tree. At each stage, that word is compared to the word already stored at the node, and is percolated down to either the left or right. Eventually the word either matches something already in the tree (in which case the count is incremented), or it has to be added at the edge. If a null pointer is encountered, a node must be created and added to the tree.

Storage for the new node is fetched by a routine **talloc**, which is an obvious adaptation of the **alloc** we wrote earlier. The new word is copied to a hidden place provided by **alloc**, the count initialized, and the two children made null. This part of the code is executed only at the edge of the tree, when a new node is being added. We have (unwisely for a production program) omitted error checking on the values returned by **alloc** and **talloc**.

If the word has been seen before, it will be found in the interior of the tree. In that case its count is incremented. Otherwise, **tree** is called recursively to place

the word in either the left or right subtree as appropriate.

treeprint prints the tree in left subtree order; at each node, it prints the left subtree (all the words less than this word), then the word itself, then the right subtree (all the words greater). If you feel shaky about recursion, draw yourself a tree and print it with **treeprint**; it's one of the cleanest recursive routines you can find.

6.6 Table Lookup

In this section we will write the innards of a table-lookup package as an illustration of more aspects of structures. This code is typical of what might be found at the heart of a macro processor or compiler for symbol table management.

There are two major routines and a table. **install(s, t)** installs the string **s** into the table with the string value **t**. **lookup(s)** searches for **s** in the table; if found, it returns a pointer to the place in the table where it was found.

The algorithm used is a hash table search — the incoming name is converted into a small integer, which is then used to index into a table. each table entry is the beginning of a list of values that have that hash value.

Here is the code.

```
#define     HASHSIZE   199   /* must be prime */

struct nlist { /* basic table entry */
        char  *name;
        char  *def;
        struct nlist *next;   /* next entry in the chain */
};

struct nlist   *hshtab[HASHSIZE];       /* the table of pointers */

struct nlist *lookup(str)    /* look for str in hshtab */
char *str;
{
        register char *s1;
        register struct nlist *np;
        static struct nlist nodef;   /* return this if not found */

        s1 = str;
        for (hshval = 0; *s1; )
                hshval += *s1++;
        hshval %= HASHSIZE;
        for (np = hshtab[hshval]; np != NULL; np = np->next)
                if (strcmp(str, np->name) == 0)
                        return(np);
        return(&nodef);
}
```

lookup performs the hashing operation on the string, in this case adding up the character values and forming the remainder modulo the table size. This produces a starting index in the table **hshtab**; if the argument is to be found, it will be in the chain of entries beginning there. If **lookup** finds the netry already present, it

returns a pointer to it; if not, it returns a pointer to a structure known not to be part of the list.

install uses **lookup** to determine whether the name being installed is already present; if so, the new definition must supersede the old one. Otherwise, a completely new entry is created.

```
char *install(nam, val)      /* install (nam, val) in hshtab */
char *nam, *val;
{
        register struct nlist *np, *lookup( );
        char *strsave( );

        if ((np = lookup(nam))->name == NULL) { /* not found */
                if ((np = alloc(sizeof(*np))) == NULL)
                        return(np);
                np->name = strsave(nam);
                np->def = strsave(val);
                np->next = hshtab[hshval];
                hshtab[hshval] = np;
        } else {                /* already there */
                free(np->def);
                np->def = strsave(val);
        }
        return(np->def);
}
```

strsave merely copies the string given by its argument into a safe place, obtained by a call on **alloc**.

```
char *strsave(s)     /* save string s somewhere */
char *s;
{
        char *p, *alloc( );

        if ((p = alloc(strlen(s)+1)) != NULL)
                strcpy(p, s);
        return(p);
}
```

With this must code in hand, it's a very small step to write, for example, a small macro processor, capable of handling the **#define** statement so long as there are no arguments.

Exercise 6-2: Write a routine which will remove a name and definition from the table maintained by **lookup** and **install**. □